Seamless configuration of virtual network functions in data center provider networks

Noname manuscript No.
(will be inserted by the editor)

Seamless Configuration of Virtual Network Functions in Data Center Provider Networks

Serena Spinoso · Marco Leogrande · Fulvio Risso · Sushil Singh · Riccardo Sisto

Received: date / Accepted: date

Abstract Network Function Virtualitazion has enabled Data Center Providers to offer new service provisioning models. Through the use of data center management software (cloud managers), providers allow their tenants to customize their virtual network infrastructure, enabling them to create a network topology that includes network functions (e.g., routers, firewalls), either chosen among the natively supported catalog or provided by third-parties. In order to deploy a ready-to-go service, providers have also to take care of pushing functional configurations into each network function (e.g., IP addresses for routers and policy rules in firewalls).

This paper proposes an architecture that extends current cloud management software to enable the configuration of network functions. We propose a model-based approach that exploits the use of additional software components, i.e. translators and gateways, which are VNF-agnostic, i.e. they are vendor-neutral and not specific for a particular type of network function, and do not require any change in the VNFs. A prototype of this solution has been also implemented and tested, in order to validate our approach and evaluate its effectiveness in the configuration phase.

Keywords Network function configuration · Network Function Virtualization · VNF description

Serena Spinoso · Fulvio Risso · Riccardo Sisto
Department of Control and Computer Engineering,
Politecnico di Torino, Italy
E-mail: first.last@polito.it

Marco Leogrande · Sushil Singh
PLUMgrid, Sunnyvale, CA, USA
E-mail: first.last@plumgrid.com

1 Introduction

Data Center Providers (DCPs) have implemented new ways to deploy and manage networking services, because their old provisioning model was too strictly intertwined with the physical network topology, being based on typical switching and bridging solutions. New service provisioning models have been enabled thanks to the innovation of the Network Function Virtualization (NFV) paradigm [1], which decouples software implementations of network functions (i.e., Virtual Network Functions) from physical computing, storage and networking resources. Following this paradigm, data centers are managed through a cloud manager (CM) such as OpenStack [2], which is a software suite that handles management tasks such as the deployment of Virtual Machines (VM), the creation of the virtual network topology, and more.

Recently, DCPs have also enabled a novel tenant-centric provisioning model, where tenants can define their own virtual infrastructure [16], with the assurance of a safe multi-tenant environment. A virtual infrastructure, which is the set of components that are required to deliver a complete service to a requesting tenant, include VMs, the actual network topology (e.g., virtual links, IP networks), and VNFs (e.g., routers, switches, firewalls, NATs and other). The creation of such virtual infrastructure can be done by either choosing network functions[1] from the catalog offered by a DCP or by deploying third-parties VNFs.

Cloud managers can rely on additional systems for configuring network services and functions: examples are Puppet [5], Chef [6], Ansible [7] and others. One advantage of such solutions is the possibility to integrate a network function without any modification to the function itself, thanks to the use of additional software modules such as agents or plug-ins. This approach, in fact, relieves VNF developers from the burden of developing a special instance of their functions for each particular CM adopted by the provider. However these tools miss a high-level agility in configuring VNFs because they are targeted to very expert users (i.e., providers, administrators, developers etc.). Existing configuration systems generally have a very steep learning curve and this implies that non-expert tenants cannot build their virtual networks without learning how to use the tool allowed by their DCP. Moreover, configuration tools generally force users to create VNF configurations in the format needed by the tools themselves, without providing a high-level representation of such configuration parameters (e.g., no separation between the representation of an IP address and its real value). In this way, such tools cannot exploit the advantages provided by model-based configuration approaches, which have been recently proposed in the literature (e.g., [8] and [4]).

Model-based configuration means defining a representation (i.e., a data model) for the configuration parameters of each VNF. The configuration, defined through the above data model, is then automatically processed internally by the system, hence generating the actual configuration parameters (e.g., IP

[1] In this paper we use the terms VNF and network function interchangeably.

addresses associated to all the interfaces of a router VNF), which are then pushed into the VNF. An advantage of the separation of the actual VNF configuration from its high-level representation is that it does not force DCPs to use a single tool (e.g., Puppet) for configuring network functions. This makes VNF insertion simpler, because programmers can integrate their VNF implementations in any CM that supports the same model-based approach.

Among the recent solutions that follow this trend, an informal working group of network operators has proposed the use of vendor-neutral data models through the OpenConfig project [4]. OpenConfig aims at creating a set of YANG-based models of network functions, leaving the choice of the strategies for pushing the actual configuration, automatically derived from the data model, to the operators such as the DCP. ForCES [8] is another example of VNF-independent configuration approach, which relies on a unified model of network abstractions and makes uses of a single protocol to control the VNF lifecycle. In order to use this approach with already existing VNFs, either the VNFs have to be updated with the addition of an implementation of the protocols specified by the above standard, or new adaptation components have to be provided, in order to guarantee the seamless integration of the existing VNFs in the new architecture.

In line with this recent trend, we propose a new architecture based on vendor-neutral network function data models defined through *VNF descriptions* that (*i*) facilitates the DCP in building a rich VNF catalog by adding services that can offer a simple and uniform configuration plane to tenants, (*ii*) enables non-expert tenants to configure their network services through that simple and coherent interface, and (*iii*) offers VNF developers an easy way to integrate their services in the CM used by the DCP.

Relying on previous works [19] and inspired by advantages of the existing agent/plug-in-based solutions that avoid changes in the VNF code, we design an architecture that exploits additional *translation modules*. The distinctive features of our additional modules are that they are VNF-agnostic (differently from configuration plug-ins), while they are specific per-configuration strategy. In this paper, we define *configuration strategy* as the combination of the protocol used to send the configuration (e.g., SSH, NETCONF, SNMP, etc.) and configuration method (e.g., command line interface (CLI), REST API, file etc.) that has to be used to fully configure a VNF. In particular, the characteristic to transparently support multiple configuration strategies is a clear differentiation compared to existing solutions, which usually make use of a single protocol or interface (e.g., Puppet).

In summary, this paper examines how DCPs can enhance their CM for enabling the proposed configuration approach. In particular, this paper proposes a solution that relies on new components and a different way to input configuration data to perform all the necessary configuration tasks, and does neither require additional per-VNF control modules (such as an additional VM that provides the adaptation layer between the CM and the VNF, or an additional module running natively in the CM), nor changes in the VNF code

to integrate it in the CM, such as the the implementation of an additional configuration strategy in the VNF itself.

The remainder of the paper is organized as follows: we start with the main challenges addressed by the proposed solution during its design in Section 2 and present how this work is positioned with respect to the current literature in Section 3; we also provide an overview of the proposed solution for configuring VNFs in Section 4; Section 5 presents a possible implementation of this architecture, which was validated and tested through different use cases, as it is presented in Section 6. This paper then concludes with some considerations and possible future works in Section 7.

2 Objectives and challenges

This paper presents a flexible, scalable and VNF-agnostic solution to configure VNFs based on (*i*) a VNF-independent formalism to describe the data model of any VNF, (*ii*) a set of VNF-dependent high-level data models (based on the previous formalism) that describe the function itself, and (*iii*) a set of VNF-independent components called *translators* and *gateways* that are in charge of translating high-level configuration directives into the actual configuration commands, which are VNF-specific. In our case, network functions (and their models) can be provided dynamically to be deployed in tenant virtual networks and CMs must guarantee their complete integration, even when these functions are not known in advance to the CM (e.g., in case of third-parties VNFs). This involves allowing the communication with other components (such as other network functions) and configuring the functions themselves. In particular the problem faced by this paper is that after creating the virtual network, configuring the network paths (e.g., OpenFlow rules) and installing the chosen VNFs, tenants have to configure them, in terms of functional and behavioral information (e.g., black-listed domains for DNS filter).

The enhancements of CMs for enabling seamless configuration can bring benefits to all the actors involved, which are the DCP, VNF Programmers and Tenants.

From a DCP point of view, the use of a flexible and automated configuration approach can facilitate the insertion of new VNFs into its catalog, as the manual intervention of the DCP is no longer required each time a new VNF has to be added, with well-known consequences in terms of provisioning agility (minutes instead of weeks). This has a beneficial impact also on costs, as the adoption of VNF-independent high-level formalisms for data models reduces the difficulties in configuring different VNFs and favors the migration to industry standards such as the one proposed by the OpenConfig [4] project, based on the YANG language. Furthermore, a model-based approach enables also DCPs to enforce additional controls such as integrity checks and verification of the configuration correctness before actually deploying the requested virtual network, as proposed in some recent works [20, 21]. Another example is the implementation of an automatic reconfiguration process, such as the

realignment of configuration parameters across multiple VNFs (e.g., the IP network assigned by the DHCP and the IP subnet used by the firewall to filter incoming traffic), although this requires the development of new advanced automated tools that guarantee the correctness of the generated configuration. A possible answer to the above problem can be found in [18], which focuses on security applications and exploits refinement-based techniques to generate and deploy the proper functional configurations in the controlled VNFs. The resulting configurations are automatically derived from a set of high-level security statements, defined by the tenant itself, and are proved to be correct thanks to the mathematical background those techniques are built upon.

From a VNF programmer perspective, instead, our solution would relieve programmers from the burden of integrating their VNFs in every architecture, for example, which may require the development of CM-specific plug-ins. Hence a model-based approach can make VNFs immediately usable in any present and future DCP architectures (i.e., CMs), without the necessity of special integration efforts (e.g., reusing VNF models). In particular, our solution is designed so that VNFs can be integrated without supporting additional protocols, but exploiting those configuration strategies the function natively supports.

Finally, tenants can benefit from an enriched sets of functions, hence more services. Moreover, they are allowed to build and operate virtual networks without knowing the low-level configuration details (e.g., command line of a router or configuration files for a DNS filter) because our solution hides such technical details. In fact, DCPs could also provide a unified API (e.g., a dashboard) in order to facilitate tenants to experience a uniform way to program and configure the entire network infrastructure, including both topology information (e.g., links and VNFs) and the configuration required by the VNFs themselves. For example, a tenant could be able to configure a router through the same API that he used to deploy the function into the network.

While the advantages of having an architecture able to automatically configure VNFs according to the model-based approach are clear, we can envision two problems. First, the semantic of a configuration depends on the network function itself, as the parameters used to configure a router are clearly different from the ones needed by a firewall. This requires (*i*) VNF-specific data models, although created with a language that is VNF-independent and hence can support arbitrary functions, and (*ii*) a set of components able to dynamically understand such descriptions and apply them to the target VNF. Second, network functions may require different strategies for being configured: some support configuration methods like a web-based interface, a REST API, or an SSH-based configuration; others can be configured via SNMP, and more. This requires the translation of high-level configuration directives coming from the tenant into the specific commands available in the chosen configuration strategy (e.g., SNMP MIBs, configuration files) and the implementation of a communication protocol in charge of transferring the above configuration to the target VNF.

Finally, the target architecture should avoid the insertion of any VNF-specific configuration component inside the DCP's network, such as dedicated software modules that provide the interface between the uniform and user-friendly configuration interface exposed to the tenants and the actual configuration strategy supported by the specific VNF. In addition, the VNF images should be kept unchanged independently of the CM under consideration and the configuration strategies chosen by the DCP and/or supported by the VNFs. Finally, in case a VNF supports multiple configuration strategies, the architecture should be able to allow the DCP to choose the best one based on a cost function and/or its management policies.

3 Related Work

In this section we investigate existing approaches for configuring (third-parties) VNFs. Even though the research world has presented different works somehow related to ours, most of the examined solutions focus on a subset of the problem we face, for this reason we have grouped them in several categories.

Agent-Based Configuration Approach. Several tools have been proposed to make the configuration and installation of additional software easy in a data center. Puppet [5], Chef [6], Ansible [7] are examples of existing configuration management systems, which aim at simplifying the task of managing large and complex compute deployments and keeping the system up to date.

One advantage of this kind of solutions is that the responsibility of enabling the communication between the VNF and CM resides on the DCP side, because such tools are based on a master-agent model (like Puppet), which require the use of agents running in each node to configure and update the network services installed on it. This means that VNF Programmers do not have to implement additional software, apart from their functions, and also DCPs avoid the installation of unknown software (i.e., VNF-specific plug-ins) in their network. On the other side, such solutions generally present drawbacks like steep learning curve, no abstraction of the network function configurations and also difficulty in managing physical instances of network functions due to the installation of non-native support agents (e.g., Puppet's agent). Moreover, most of them rely on a centralized management module, which has to collect the configurations of all the functions and services installed in the network and manage them, bearing all the well-known problems of a centralized solution. On the contrary, our solution was designed to be modular and logically distributed.

Protocol-Based Configuration Approach. Among the investigated solutions for configuring network functions, SNMP [14] and NETCONF [13] are two protocols that lay on data model languages. SNMP relies on SMIv2, while NETCONF on YANG, which has been indicated in [22] as a better data modelling language compared to other languages (e.g., XML schemas). This is in line

with our implementation choices, because we have exploited YANG for implementing the network function data models.

From a DCP perspective, the use of a single configuration protocol, like SNMP or NETCONF, may limit the VNF catalog: all the non-SNMP/NETCONF functions cannot be integrated in the cloud manager in case these are the only protocols supported by the DCP. The use of a more flexible architecture that can enable more than one configuration strategy (e.g., NETCONF, RPC, REST, etc.) can relive, on one side, the VNF programmers from integrating the support of further protocols in their functions, and, on the other side, make the DCP free to chose the best configuration strategies among the available ones, based on, for example, security implications.

Model-based configuration approaches. A recent proposal in model-based configuration is represented by the open-source OpenDayLight (ODL) [3] SDN controller, which exploits a model-based service abstraction layer in order to create programmable (and configurable) network services. In particular, ODL enables the integration of third-parties VNFs thanks to the use of plug-ins in charge of the communication between the VNF and ODL, and YANG-based models that specify the data structures used and the messages supported by the northbound interface of the VNF itself.

At the best of our knowledge, the difference between our proposal and the approach taken by ODL does not consist in the model-driven abstraction, because both solutions encourage the use of models to represent data structures and primitives to generate the VNF configuration and push it down to the function itself. Instead, our solution has a different architectural design: while the SDN controller needs VNF-specific plug-ins that are developed by the VNF programmers and that have to implement an RPC API as northbound interface (i.e., toward the SDN controller), we envision the use of configuration modules that are VNF-agnostic and that can communicate with the VNF through any protocol or API (i.e., configuration strategy) supported by the DCP.

The ForCES framework [12], defined by the IETF Forwarding and Control Element Separation working group, is another example of model-based configuration approach and it addresses the creation, configuration, and resource assignment of VNFs, exploiting an object-oriented model. In this context, [8] argues for the need of a unifying common network abstraction model for both forwarding aspects and network functions. This model is processed by a Network Function Manager in charge of accessing to each device through appropriate APIs and managing their life cycle. The authors have also provided a proof-of-concept of the ForCES applicability in an NFV architecture [9].

However, instead of exploiting existing configuration strategies already supported by a network function, as it is envisioned by our work, their solution uses an additional protocol (namely ForCES) for configuration and management purposes. Moreover, the ForCES framework was designed for configuring network datapaths by means of an XML schema: our solution, on the contrary,

aims at defining function models for configuration purposes, without considering network connections and resources and use YANG as data modelling language, instead of XML schemas, for the aforementioned reasons.

OpenConfig [4] is an Industry-based working group that proposes another model-based approach for configuration and management and is currently building a public database of vendor-neutral data models of network functions, created using the YANG language. The development of these vendor-neutral data models can facilitate VNF integration, because, it focuses on making function model natively supported on networking hardware and software platforms reusable for any DCP network. The main goal of the OpenConfig project is concentrated on the modeling phase, leaving DCPs free to implement any strategy for pushing configurations into network functions. Our main objective is instead to define all the features required by a cloud manager to handle all the steps needed for a complete configuration process.

Other Approaches. Finally, it is worth summarizing how some of the most recent orchestration architectures proposed in the literature refer to the VNF configuration problem, starting with the well-known ETSI-driven NFV architecture [1]. Even though the ETSI NFV project has considered the problem of configuring VNFs and has proposed the Management and Orchestrator component (MANO) to take care of management and configuration tasks in the NFV architecture, only few works (e.g., [18]) are currently available that investigate this issue.

An example of MANO solution is vConductor, presented by Shen *et al.* [16], which enables users to define their virtual networks. The authors have designed their solution to exploit OpenStack [2] network and compute resource provisioning frameworks, providing a proof of concept. In particular their prototype exploits a data model for service, computing and networking aspects, neglecting configuration parameters and data as we, instead, envision.

Another important architecture has been developed within the FP7 project UNIFY [15], which aims at orchestrating any VNFs available in the whole network of the telecom operator, which addresses the configuration problem by defining a dedicated module as responsible of this task. A proof of concept prototype of this architecture is available through the ESCAPE framework [17], which deploys virtual networks by processing a data model, named Network Function-Forwarding Graph (NF-FG). However, the problem of the configuration has not been properly investigated, as the NF-FG model includes only basic parameters such as IP addresses and cannot be seen as an acceptable answer to the very general problem of VNF (and service) configuration.

4 Architecture

This section presents first the high-level overview of the whole architecture, then it will show a more detailed view of the components in charge of con-

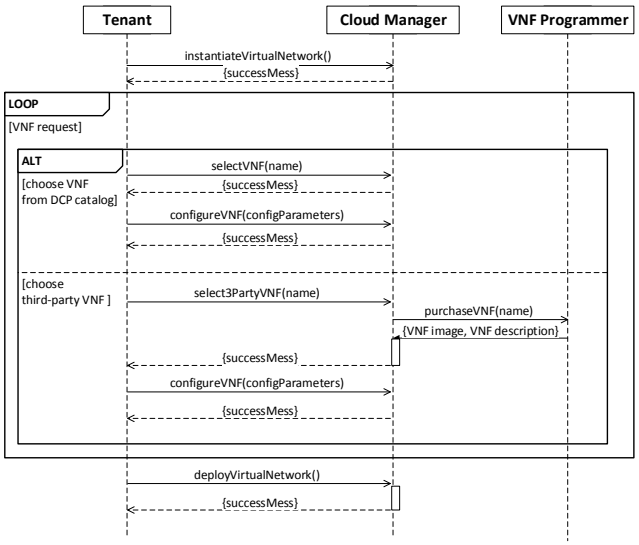

Fig. 1: Interaction between different actors.

figuring VNFs and the inputs they require for their tasks, and the respective actors that are in charge of that data.

4.1 Architecture overview

In order to perform a complete service deployment, including the configuration of the VNF, the CM needs both the VNF image and its data model, which is collected in a *VNF description* and must be provided by the VNF programmer. The DCP has to store both the VNF image and description in the proper modules of its CM, in order to have them available when the tenant issues a service request (Figure 1). At this point, the tenant can configure its VNFs through a VNF-agnostic interface provided by the CM (e.g., REST API, dashboard, etc.). Finally, the configuration module will automatically generate and push the actual configuration in the VNFs, making the requested service fully operative.

In this process it is important that the VNF description is defined in a unified format in order to help mainly programmers and DCPs. The VNF programmer can define the main functional information (e.g., firewall policy) and the configuration protocols and methods (i.e., configuration strategies) for pushing them into the VNF in a way that is recognized by any DCP, which, in turn, are able to add to their catalog and use any VNF that adheres to the unified description format. In order for this to be possible, DCPs must be able to configure any VNF regardless of their intrinsic details.

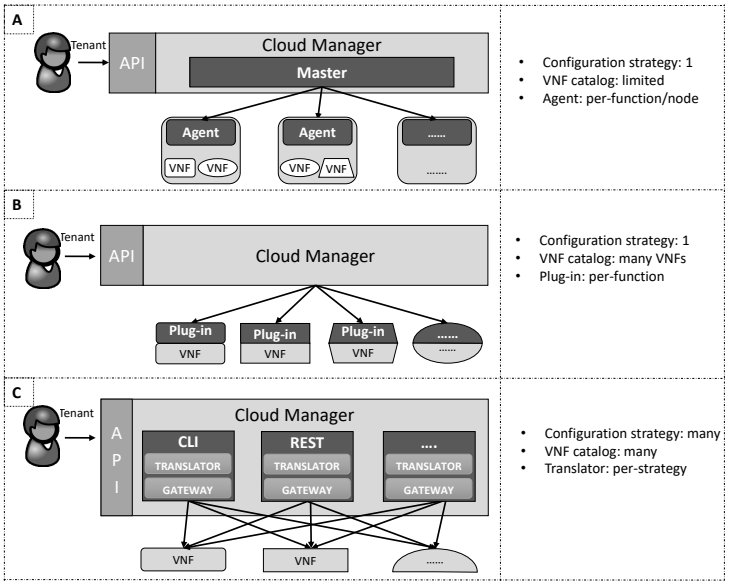

Fig. 2: Possible configuration-oriented CM architectures.

In order to enable this kind of configuration service, many features are needed in a CM, in particular the integration of modules that perform the configuration and the modification of the exposed interface to receive the VNF configuration parameters.

A possible configuration-oriented architecture of a CM is shown in Figure 2 (*case A*): a master component is in charge of generating the VNF configurations, while the agents are software modules installed in the network where the VNFs are deployed, that monitor if the VNFs need some configuration updates and, in case alert the master.

This is the approach adopted by solutions like Chef, Puppets, etc., where the master/agent communication is based on HTTP API. This type of solution avoids the necessity to develop additional code (i.e., adaptation of the VNFs to the configuration protocol chosen by the DCP, or the implementation of VNF-specific plug-ins), hence facilitating the operations of the VNF programmer. However, this architecture presents some limitations *(i)* with respect to the VNF catalog, because functions that cannot be configured through the agent cannot be integrated in the system (or further effort must be spent for their integration), and *(ii)* it requires the presence of an agent that runs aside each VNFs.

Another possible architecture is depicted in *case B* (Figure 2), where the configuration engine is not centralized inside the CM, but it is moved to VNF-specific plug-ins. This is the approach currently adopted by OpenDayLight, which requires a control plug-in developed by the VNF programmer to enable the communication between VNF and cloud manager by means of, for example, an RPC API. From a DCP point of view, this solution leads to: *(i)* decreased

VNF integration cost than *case A*; *(ii)* a richer set of offered virtual services thanks to the simplicity in integrating new VNFs; *(iii)* the necessity to execute control modules that are developed by third-parties, which might introduce additional security issues.

Our solution is represented by the architecture shown in *case C* (Figure 2), which translates the high-level configuration parameters in the actual VNF configuration commands and pushes them into the function. This solution (*i*) avoids any change in the VNF source code and/or the implementation of additional configuration plug-ins, (*ii*) avoids the installation of control agents in the network in order to better scale with the number of deployed VNFs and, finally, (*iii*) supports multiple configuration strategies (e.g., CLI, REST, etc.).

In particular, our approach is based on the splitting the configuration engine in two orthogonal (and sequential) tasks, which consists in the translation of high-level configuration parameters into a particular format required by a VNF and their delivery to the function. Thus, a *translator* takes care of the first task and a *gateway* will take care of the latter, transferring and installing the VNF configuration by using one of the configuration strategies already supported by the function itself.

This logically distributed architecture allows providers to optimize the configuration task by instantiating a variable number of translators and gateways, possibly only upon request, based on the current load of the system and the number of VNFs that have to be configured in order to implement the overall service request.

The use of translators and gateways allows the system to increase the number of supported configuration strategies (hence, VNFs that require unconventional configuration methods and protocols) without impacting on the existing ones, which continue to operate as usual. Moreover, these modules exploit the data model descriptions of the VNFs for which the configuration has to be created, enabling DCPs to support an unlimited number of network functions. Also the separation of translating VNF configuration from the task of delivering it to the VNF allows to split the inputs needed by the new components. In particular, translators and gateways will use different parts of the VNF description, which are: *VNF object model*, *translation rules* and *access parameters*. The next sections will describe in detail these inputs and how they are exploited by the new components.

4.2 Configuration translators

Since each configuration strategy has its own peculiarities (e.g., for a CLI-based configuration, it is necessary to know the commands for enabling administrative authorization), the architecture includes a *configuration translator* for each configuration strategy the DCP wants to support (Figure 3). A translator hence must be aware of all the particular techniques and quirks needed for the strategy it is in charge of.

The use of separated translators makes also the system more extensible and manageable, as it allows an easier insertion, replacement and removal of supported configuration strategies: when the DCP wants to support a new strategy, he has just to make a new translator available (and, in turn, a new gateway). Furthermore, the system becomes scalable with respect to the number of VNFs running in the system: one translator (and gateway) enables the integration of a number of network functions that support that strategy. The larger the number of VNFs, the larger the number of translators and gateways that are instantiated. This solution contrasts with the limited scalability of agent-based architectures where a single agent may become a bottleneck.

Translator inputs. In order to perform its job, a translator needs the *VNF Object Model* (OM), which is the VNF-specific model that we exploit to represent the function data inside the system (*point 1* in Figure 3). In particular a VNF object model represents a description of the data-structure instantiated for storing the configuration parameters of a VNF. This means that each VNF must be associated with its object model in order to be correctly integrated into the system. Note that more than one OM may be necessary for the same type of VNF. For example, a firewall from a first manufacturer can support features that are different from the ones supported by a firewall of another manufacturer, requiring further configuration parameters for such additional features and hence a different object model.

Moreover since the object model is only the specification of the data structure used to store configuration parameters, an instance of the OM for each deployed VNF is stored inside the CM, namely the *VNF Object Model instance* (OM instance). For instance, we can consider the VNF OM as a class declaration in an object-oriented programming language, while the VNF OM instance can be seen as a particular instance of that class. In particular, when the CM receives the request to deploy a new set of services (*point 2*), for each VNF that is being instantiated a specific OM instance is created (*point 3*). Referring to the previous example, if the tenant has required two VNFs of the same firewall, two OM instances are created from the object model of that particular firewall, one for each firewall VNF deployed in the network. Each OM instance will contain the set of policy rules configured for its associated firewall.

Among the other aforementioned advantages, the use of data models makes any changes in DCP-provided API easier and transparent for the internal processes of the system. This avoids also the use of data-structure formats for collecting VNF configurations that would be translator-specific.

Another input coming from the CM is a set of *translation rules* (*point 4*), i.e. directives used to drive the translator in generating the configuration of the VNF in the right format. They express the way to translate the structure and content of the OM instance into the specific structure/format required by the VNF. Referring to the previous example of the firewall, translation rules specify the format of policy rules according to the specific firewall in use.

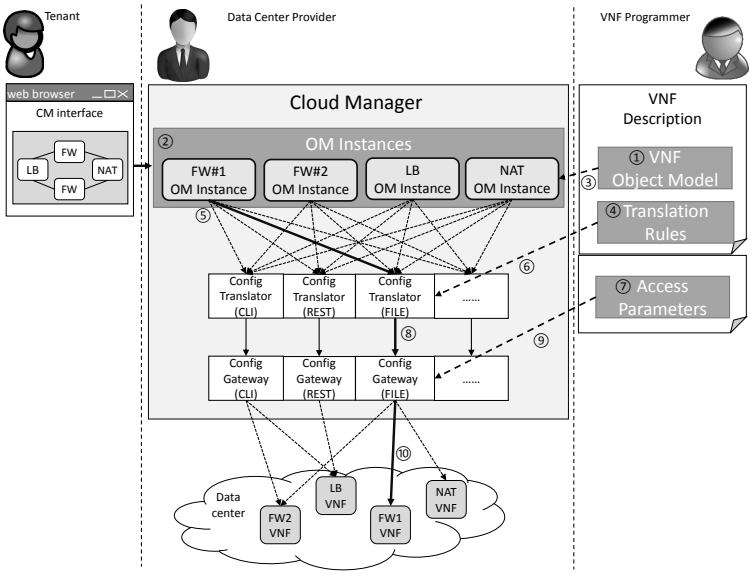

Fig. 3: Overview of a CM architecture for configuring VNFs.

The particular format used to configure a VNF depends also on the configuration strategy supported by the function itself (e.g., CLI, REST API, file etc.). For instance, a configuration through REST interface has certainly a different format from one used for CLI-based configuration.

To better clarify the idea under the translation rules, let us consider the previous example of the firewall and suppose that a policy rule can be set through a command line like "*add rule -source 130.192.31.24 -destination 8.8.8.8 -action ACCEPT*": an example of translation rule may be like "*add rule -source IP_VALUE -destination IP_VALUE -action ACTION_VALUE*", where the actual configuration parameters values (i.e., IP address and action) are stored in the OM instance of that firewall. Further details about the format of the translation rules in our solution are presented later.

As well as the object model, translation rules are both configuration strategy- and VNF-specific, since each network function has its own primitives to be used in the configuration phase. Hence both the VNF object model and translation rules can be provided by the programmer through the VNF description.

With respect to which inputs the configuration translators need, the new modules have been designed to receive: *(i)* configuration parameters saved into the OM instance of the VNF (*point 5* in Figure 3); *(ii)* translation rules for deploying such parameters into the VNF in the right format (*point 6*).

It is interesting to note that an OM instance is self-descriptive and hence translators can discover the structure of an OM from any instance of that model.

4.3 Configuration gateways

The proposed architecture includes also a *configuration gateway* for each translator and, in turn, for each strategy supported (Figure 3). Gateways are in charge of delivering the actual VNF configuration into the function by means of the configuration strategy for which it is authorized. In order to achieve this goal, a gateway needs surely the result of the configuration translation process, which represents the final configuration of the function (*point 8* in Figure 3). However a configuration gateway requires another input to perform its goal, namely the *access parameters*.

Gateway input. The *access parameters* (*point 7* in Figure 3) are used to instruct the system on how to contact the VNF and update its configuration, in order to complete the configuration of the VNF itself. Examples of access parameters are IP addresses, port numbers, administrative credentials, commands for entering in configuration mode and everything that describes how to add the policy rules inside the firewall we have considered before.

In our vision access parameters should be standardized for each configuration strategy, because each strategy needs different information: for example, in a configuration though files, CMs must know the path where configuration files are stored. DCPs and programmers can then set the actual values of those parameters: DCPs would establish the management information internal to his architecture (e.g., IP addresses of control interfaces), while programmers would set parameters related to the internal mechanism of the VNF (e.g., root credentials). This implies that access parameters are strategy-specific, but some of them are also VNF-independent. This is the reason why we do not include such parameters into VNF descriptions. Further details on how access parameters are stored in a real implementation of the architecture are provided later.

5 Proof-of concept implementation

This section describes a proof-of-concept implementation of the proposed architecture. We start by presenting some details that have not been included in the previous description in order to keep the architecture description more generic (i.e., inputs format and languages). We then continue with a description of our prototype.

5.1 Object Model

The VNF Object Model is based on the YANG data modeling language [10], developed by IETF and extended for our purposes. YANG has been designed to model configuration data and state, which can be manipulated through a protocol such as NETCONF. YANG was chosen because of it is protocol-agnostic,

implementation-independent and human-readable. YANG is also easy to extend with new directives without impacting the compatibility with previous implementations. Furthermore it is oriented to network configuration tasks, hence it provides an excellent foundation for our problem as well.

This language offers also a wide set of directives to validate its statements. Examples are type checking, default values, mandatory/optional statements and their cardinality, value ranges checking, and other. While other simple validations are possible through the definition of new YANG types, more complex validations (e.g., dependency checking between statements) would require new extensions. The support for validating primitives could allow VNF programmers to include directives that can be checked against configuration parameters provided by the tenant. However in this paper we are interested in the configuration process and prefer to leave the checking and verification of configuration correctness as future work. Hence, in our implementation YANG has been exploited to define the VNF object model, which includes the most significant data structures that are required to properly configure the function.

While the YANG language provides the set of advantages listed before, our architecture can be implemented with any other language that present similar characteristics; a possible alternative to YANG is represented by the XML Schema. In this respect, XML Schema is more mature and already well standardized, but it is more verbose, as shown by comparing the same data structure defined in XML Schema (Listing 7 in Appendix A), where we have defined the same data structure shown in Figure 1. In addition, YANG is being adopted by different projects in the field of network management such as OpenConfig, and new software artifacts such as the OpenDaylight SDN controller, hence it should be more familiar at lest to the network managers.

An example of a possible YANG Object Model (i.e., VNF description) is shown in Listing 1, where we define a structure to describe the state of the Ethernet interfaces of a router. The idea is to have a data structure to enumerate all the interfaces of a router (the top-level `interfaces` list) and, for each of them, store all their network and physical addresses[2] (respectively the leafs `address` and `hwid` in the nested `ethernet` list).

In our solution, the YANG VNF description file includes also translation rules (presented in the next section), which are VNF-specific.

5.2 Translation rules

As shown in Listing 1, translation rules take the form of special comments in the YANG-based VNF description, using the following structure:

```
//ConfigTransl:<Transl_N>:<Rule_N> <Rule_V>
```

[2] Usually a network interface is assigned only one network and physical address, but this is not true in the general case.

```
module router {

  import ietf-inet-types { prefix inet; }
  import ietf-yang-types { prefix yang; }

  ...

  list interfaces {

    //ConfigTransl:file:header "//Start Interface List\n";}
    //ConfigTransl:file:list_format "%NAME {\n";
    //ConfigTransl:file:separators "\n}\n";
    //ConfigTransl:file:footer "}\n//End Interface List";

    key name;
    leaf name {
      type string; }

    list ethernet {

      //ConfigTransl:file:list_format "%NAME %VALUE {\n";
      //ConfigTransl:file:separators "\n";
      //ConfigTransl:file:footer "}\n";

      key name;
      leaf name {
        type string; }

      leaf address {
        //ConfigTransl:file:leaf_format "%NAME %VALUE\n";
        type inet:ipv4-address; }

      leaf hwid {
        //ConfigTransl:file:leaf_format "hw-id %VALUE\n";
        type yang:mac-address;}
      }
   }
}
```

Listing 1: YANG language example: an excerpt of a router VNF description.

where <`Transl_N`> specifies which configuration translator (and, in turn, configuration strategy) the rule belongs to and can assume values like "`file`", "`cli`", "`rest`", etc.. Instead, <`Rule_N`> and <`Rule_V`> represent the rule name and value, interpreted as strings. This structure allows to group all the rules for a given translator under a specific prefix, in a way that is similar to the concept of the *namespace*. This permits the presence of multiple translation rules in the same YANG file, which can be useful when the VNF can support different configuration strategies.

The example presented in Listing 1, which assumes that the router is configured through a file-based translator, shows some translation rules that create the properly formatted output, which are: *(i)* `header` and `footer` are inserted respectively before and after the current YANG element (e.g., `list` or `leaf`) when generating the final configuration; *(ii)* `separators` is used to divide child nodes of the current statement; *(iii)* `list_format` and `leaf_format` work like a `printf` of the C language, in which `%NAME` and `%VALUE` are expanded with values depending on the context. In particular, `%NAME` and `%VALUE` represent respectively the name of their YANG statement (e.g., "ethernet" for the `list ethernet` and "address" for the `leaf address`) and its actual value (in the case of a `list`, it will be the value of its key).

Although other configuration strategies may need additional (or different) information such as the exact ordering sequence of the commands to be issued in a CLI-based configuration, this does not represent a problem, as new translation rules can be defined with the format needed by the specific translator. Furthermore, we could leverage hierarchical data structures, which are natively offered by YANG. For instance, the current implementation serializes the YANG Object Model of a VNF, hence assigning a lower priority to the nested elements than their root statement.

None of the keywords is mandatory: an extreme case, thus, is a YANG statement that does not have any translation rule. In this case that node will not appear in the configuration output.

5.3 Access parameters

In general, access parameter are VNF-independent, but depend on the specific configuration strategy chosen by the VNF (e.g., a network-based configuration requires the TCP port to connect to, while a file-based configuration requires to know where that file is located). Hence the DCP has to define the proper set of access parameters for each supported configuration strategy. This is the reason why in our solution the above parameters are not included in the VNF description, but they are stored in another object that is used only by the CM and may not be fully exported to the tenant. In order to simplify the deployment, also access parameters are described using the YANG language.

Moreover, a new OM instance for the access parameters is automatically created when a VNF is deployed and associated to the function, because this instance must store the actual values of access parameters for loading a new configuration into that VNF. The access parameter OM instance is also associated to its function thanks to the `name` field (Listing 2), which contains the VNF identifier inside the system.

An example of OM of access parameters for file-based configuration is shown in Listing 2. Here VNF programmers, even tenants, must be able to set the access parameters related to the VNF only. In other words, they must not have the privileges for setting parameters like IP addresses of management interfaces and others. An example of possible OM instance associated with

```
module ConfigTransl2File {
    list access_param {
      key name;
      leaf name { type string;  }
      leaf ip_address { type string;  }
      leaf port { type string; }
      leaf user_name{ type string; }
      leaf user_key { type string; }
      leaf commands { type string; }
      leaf file_name { type string; }
      leaf file_path { type string; }
    }
}
```

Listing 2: Excerpt of access parameter object model.

the aforementioned model and related to a router VNF is shown in Listing 3, written in a JSON-like format.

The choice of the YANG language for describing the access parameters was taken also because most of the parameters we need to describe are simple (like IPv4/IPv6 address, configuration file name and path, etc.) and natively supported by YANG. In any case, if needed, we can leverage the additional YANG types defined by IETF in [11].

Finally to recap the configuration process, Figure 4 shows a detailed view of the whole architecture including all the inputs. The VNF programmer has provided both the VNF image (*point 0*) to launch the function instance, and the other inputs required by the system. In particular, the VNF object model (*point 1*) allows to build automatically the CM interface (dotted line) and OM instance associated to that function (dashed line). Translation rules are instead sent to the translators (*point 2*), while access parameters (*point 3*) are stored into a gateway-specific OM instance (its structure is defined by the DCP,

```
{  "access_param": {
     "name": "Router_94",
     "ip_address": "130.192.31.94",
     "port": "2001",
     "user_name": "router_admin",
     "user_key": "admin",
     "commands": "load /configuration/config.boot",
     "file_name": "config.boot",
     "file_path": "/configuration"
   }
}
```

Listing 3: Possible content of an access parameter OM instance.

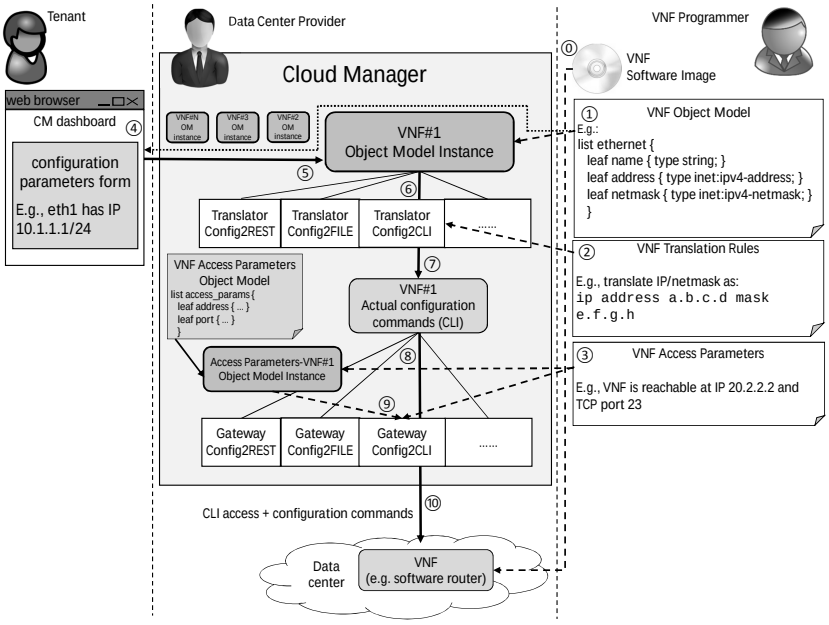

Fig. 4: Detailed overview of the enhancements in CM architecture.

as mentioned before). The OM instance associated to the access parameters is automatically created when a new VNF is deployed: this instance is then associated to the VNF and used by the gateway later on.

Through the CM interface (e.g., the DCP web dashboard in Figure 4), tenants can set the VNF configuration parameters (*point 4*), which are stored into the OM instance of the function (*point 5*). After that, this instance (*point 6*) is passed to the translator selected based on the preferred configuration strategy (i.e., CLI-based translator in the example). The combination of this input and the translation rules is used to generate the actual VNF configuration (*point 7*). Finally, the configuration gateway can retrieve both access parameters (*point 9*) and the produced configuration (i.e., translator output - point *8*) to complete the VNF configuration process (*point 10*).

5.4 ConfigTransl2File Prototype

Having defined the language and formats of the additional inputs required by the new components, we have also implemented a prototype for validating and testing the effectiveness of our solution. In our prototype, a C++ library, namely `ConfigTranslLib`, has been designed to support several configuration strategies. We have implemented a translator/gateway prototype, namely

`ConfigTransl2File`, to configure VNFs by means of files, regardless of their format (e.g., XML, text or more).

This translator receives inputs through a REST interface exposed by the CM, which are: (*i*) YANG OM instances of VNFs, where the translator can retrieve the configuration parameters chosen by tenant and the configuration file structure required by the function; (*ii*) translation rules, which have been stored in the VNF OM instance as well as the configuration parameters.

For the sake of simplicity, in our implementation the whole set of access parameters is configurable through the REST API. However, in a real vendor's implementation, just a subset of those parameters must be exposed and made public to tenants and programmers. The access parameters are then stored into another OM instance, specific for the `ConfigTransl2File` translator.

Finally it is worth noting that our solution is able to support VNFs that could require multiple configuration files. The `ConfigTransl2File` library can be instructed to write different portions of the same YANG OM into different configuration files, so that VNFs that require it can dump different parts of their data into different locations. This can be done because of the object model abstraction: the root element of a YANG module, for example a YANG list, has no difference from a nested YANG statement (e.g., container, leaf-list, list) under it, then these two elements of an object model can be the entry points associated to different configuration files of the same function.

6 Validation and testing

We validated our architecture by implementing the components required to configure two VNFs, Bind9 and Vyatta Core, respectively a DNS server and a software router, which represent two well-known, albeit very different, network functions. We decided to benchmark the performance of our prototype using one VNF at a time, omitting the case in which multiple VNFs are deployed (hence, need to be configured) at the same time. In fact, our current proof-of-concept prototype handles the two VNFs sequentially, hence requiring a total time for the configuration that is the sum of the individual components. However, it is trivial to implement the architecture with multiple gateways and translators, all running in parallel, hence achieving a configuration time that is independent from the number of VNFs that have to be configured.

Starting with the validation phase, in particular concerning the DNS server, we have defined the YANG-based description for Bind9. An excerpt is shown in Listing 4, while Listing 5 is the corresponding part of the Bind9 configuration file, generated by our prototype. As shown in Listing 5, we have configured Bind9 to act as Secondary Master (i.e., it gets the zone data from the Primary Master for that zone). Our validation methodology consists in sending configuration requests to our CM through its REST interface that aim at setting the Bind9 configuration parameters; the call triggers the `ConfigTransl2File` translator, which generates the above-mentioned configuration file. Another

REST call is then issued to initialize the Bind9 access parameters, which are stored in the proper OM instance.

Having all of the required inputs, the system is able to push the final configuration file into the VNF and restart it. The successful deployment of the configuration is validated by interrogating the Bind9 instance and checking that the returned answer are coherent with the desired configuration.

A similar validation has been performed also for the second VNF, which involves the Vyatta Core router. An excerpt of its YANG description file is shown in Listing 1. In this case we have checked that the Vyatta instance is actually configured with the desired data by checking the reachability of the IP addresses on the interfaces and its static routes. Listing 6 shows an excerpt of the Vyatta configuration file that was correctly generated by the ConfigTransl2File prototype from the description shown in Listing 1.

6.1 Testing results

Two metrics have been considered for evaluating the effectiveness of the proposed solution, which are (*i*) the elapsed time for generating configuration files and (*ii*) the reduction of complexity from a tenant prospective, which can be

```
module bind9 {

  list zone {

    //ConfigTransl:file:list_format "%NAME \"%VALUE\" {\n";
    //ConfigTransl:file:separators ";\n";
    //ConfigTransl:file:footer "};\n ";

    key name;
    leaf name {
      type string;  }

    leaf type {
      //ConfigTransl:file:leaf_format "%NAME %VALUE\n";
      type string; }

    leaf file {
      //ConfigTransl:file:leaf_format "%NAME \" %VALUE\"\n";
      type string; }

    leaf master {
      //ConfigTransl:file:leaf_format "%NAME { %VALUE; };\n";
      type string; }
  }
}
```

Listing 4: An excerpt of the Bind9 YANG description file.

```
zone ``example.com" {
   type slave;
   file ``db.example.com";
   masters { 192.168.1.10; };
}
```

Listing 5: Excerpt of the generated Bind9 configuration file.

translated in the size (i.e., verbosity) of the generated file compared to the corresponding YANG source.

Starting with the first metric, Figure 5 plots the required time for generating the configuration file versus the size of such file. We have performed multiple test runs (i.e., about 100 executions per file dimension) for both Vyatta Core (square points in figure) and Bind9 (circle points).

As show in the graphs, we have obtained satisfactory trends, because, as we expect, the time required by ConfigTransl2File grows proportionally to the size of the configuration file. This means that our solution is able to handle configurations with a growing complexity, without requiring an exponential time increase. This is in line with the constraints of assuring a good experience to tenants.

In addition, our tests demonstrate that, for real case scenarios, the time required to obtain the configuration file is on the order of tens of milliseconds. This result is also in line with the configuration times that are achieved with other agent-based solutions. In particular, we have identified Ansible [7], as one possible solution used to compare our approach. Ansible is based on agents that exploit the SSH protocol for their interaction with the VNF, which is usually supported by most VNFs. Furthermore, similarly to our approach,

```
//Start Interface List
...
interfaces {
  ethernet eth0 {
  address 130.192.31.94
  duplex auto
  hw-id 00:0c:29:64:66:1c
  mtu 1500
  smp_affinity auto
  speed auto
  }
}
...
//End Interface List
```

Listing 6: Excerpt of the Vyatta configuration file.

it avoids the installation of new agents in the VNF, hence preserving the original network function image. Compared to the configuration time needed by Ansible to push a configuration in both Bind9 and Vyatta, our solution performs slightly slower, but it never exceeds 40% the value obtained by Ansible, which is completely acceptable for humans who hardly notice this difference. Furthermore this result has been obtained with proof-of-concept code, which can be optimized in the future.

The two graphs, shown in Figure 5, report the 95% confidential interval and show that our solution takes less than 30ms in average in both use cases. The configuration time achieved by our prototype has a negligible impact on the total deployment time, which is usually on the order of tens of seconds when virtual machines have to be started. Hence, these results demonstrate that the introduction of our solution in the CM does not increase the service provisioning time experienced by tenants.

The second test suite aims at evaluating the reduction of complexity in configuring networks from a tenant prospective, achieved thanks to our prototype. In particular our solution allow the creation of YANG files in which only the main configuration parameters are exported, such as policy rules in a firewall VNF, avoiding all the details required by the specific configuration method and the possible syntactical rules (and keywords) required by the VNF native configuration method (e.g., firewall rule format, priority commands, special directives etc.).

As a metric to measure the complexity of the configuration, we used the size (in bytes) of the configuration files generated by our tool (dark grey bars in Figures 6 and 7), which represent the complexity of the native configuration method of the VNF. The above value has been compared with the size of the configuration messages that have been generated to push the configuration in the CM through our configuration REST APIs, which can be seen as the

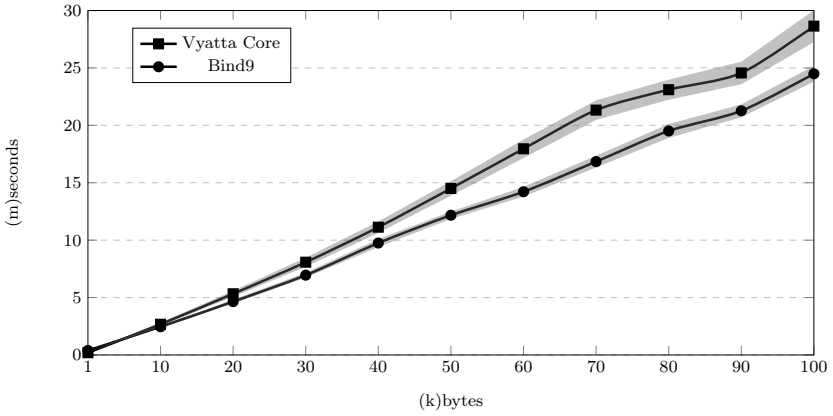

Fig. 5: Elapsed time for generating Vyatta Core and Bind9 configuration files, with 95% confidential intervals.

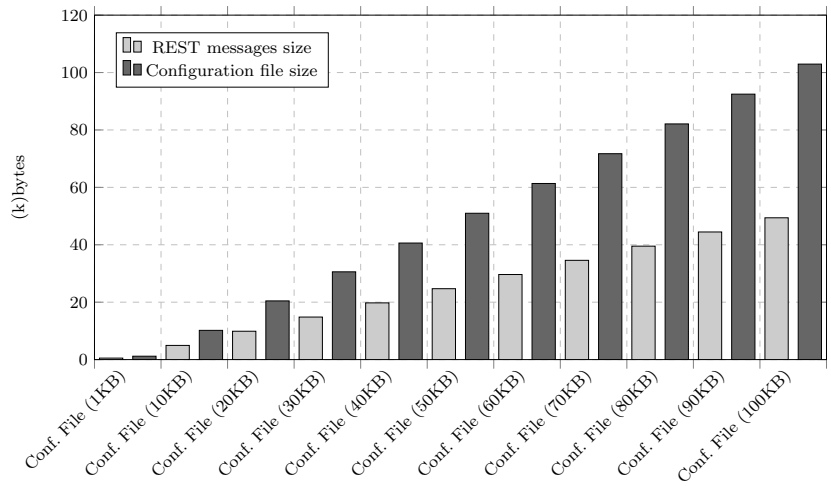

Fig. 6: Bind9 use case: reduction of configuration complexity.

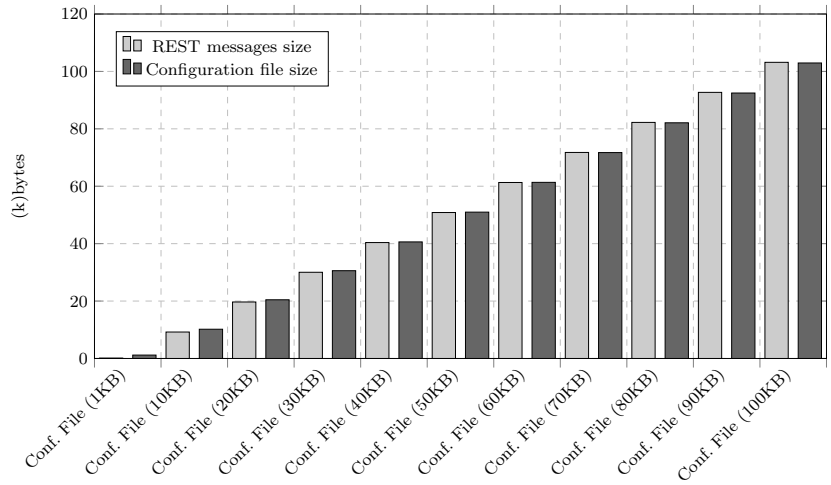

Fig. 7: Vyatta use case: reduction of configuration complexity.

size of the same configuration with our approach (light grey bars in Figures 6 and 7). In particular, Figure 6 shows the results achieved in the Bind9 case, while Figure 7 depicts the case of the Vyatta Core router.

In the Bind9 case, the reduction of complexity, that is the difference between the size of the configuration file and REST messages, grows linearly, suggesting that tenants are facilitated in configuration phase. In the Vyatta case, the size of the configuration file is approximately equal to the REST message size, hence suggesting a similar complexity. However, it is worth noting

that, with our approach, tenants are relieved from the burden of having a deep knowledge of how to configure their VNFs, since they must interact only with the CM, using a uniform configuration model across all VNFs. In addition, this result must not be attributed to a possible inefficiency of our solution as it is due to the differences existing between the two VNFs: each VNF implementation has its own configuration peculiarities and one function can be simpler than other in configuration phase. This result highlights the importance of supporting multiple configuration strategies and, in turn, multiple translators/gateways. For example, a configuration through the CLI may reduce significantly the configuration complexity of the Vyatta case, which means that the difference between the REST message payloads and the produced configuration is more evident than the configuration through file. Supporting many translators, the DCP may select the most suitable one for configuring a specific VNF instance, based on their internal management policies and costs.

Concluding, our solution reduces the effort spent by tenants in configuring their virtual services, with a negligible impact in terms of configuration time and, likely, with a simplified configuration interface.

7 Conclusion

This paper proposes a solution for one of the weaknesses of the current cloud managers used by Data Center Providers: the possibility to configure the network functions using a simple and uniform configuration method, without at the same time forcing the DCP to deploy additional per-VNF software modules or the VNF programmer to adapt its code to the configuration tools chosen by the DCP.

This paper presents a model-based approach to solve the above problem, which enables to configure VNFs in terms of functional parameters (e.g., IP address for a router and policy rules for a firewall), bringing multiple advantages for all the actors involved (i.e., DCP, VNF programmer and tenant). The cost and complexity reduction of integrating further VNFs is an example of a possible advantage for the DCP and the VNF programmer.

The proposed enhancements consist in adding new VNF-agnostic components in charge of configuring VNFs, with the associated inputs (and formats). In particular, configuration translators and gateways are designed such that: *(i)* CMs can support arbitrary VNFs without the use of specific control plug-ins or agents; and *(ii)* VNF programmers can integrate any of their functions without supporting a CM-specific configuration strategy or developing further implementation of their functions per DCP.

The proposed solution was validated and tested through two use cases of VNFs: a software router (Vyatta Core) and a DNS server (Bind9). The achieved results prove that our solution does not impact the service-provisioning time and simplifies the tenants interaction with the CM, because tenants are relieved from the burden of having a deep knowledge of how to configure each of their functions.

Possible future extensions could include further services provided by DCPs for verifying the correctness of the configuration generated and validating the correct integration of the desired configuration in the CM.

References

1. European Telecommunications Standards Institute, "Network function virtualization; Architectural Framework," October 2013.
2. "OpenStack," https://www.openstack.org/.
3. "OpenDayLight," https://www.opendaylight.org/.
4. "OpenConfig," https://www.openconfig.net/.
5. "Puppet," https://puppetlabs.com/.
6. "Chef," https://www.chef.io/.
7. "Ansible," http://www.ansible.com/.
8. E. Haleplidis, J. Hadi Salim, S. Denazis, and O. Koufopavlou, "Towards a network abstraction model for sdn," *J. Netw. Syst. Manage.*, vol. 23, no. 2, pp. 309–327, Apr. 2015. [Online]. Available: http://dx.doi.org/10.1007/s10922-014-9319-3
9. E. Haleplidis, S. Denazis, O. Koufopavlou, D. Lopez, D. Joachimpillai, J. Martin, J. H. Salim, and K. Pentikousis, "Forces applicability to sdn-enhanced nfv," in *Software Defined Networks (EWSDN), 2014 Third European Workshop on*. IEEE, 2014, pp. 43–48.
10. M. Bjorklund, "YANG - A data modeling language for the Network Configuration Protocol (NETCONF)," Internet Requests for Comments, RFC Editor, RFC 6020, October 2010. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6020.txt
11. J. Schoenwaelder, "Common YANG Data Type," Internet Requests for Comments, RFC Editor, RFC 6991, July 2013. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6991.txt
12. T. A. R. G. L. Yang, R. Dantu, "Forwarding and Control Element Separation (ForCES) Framework," Internet Requests for Comments, RFC Editor, RFC 3746, April 2004. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3746.txt
13. R. Enns, M. Bjorklund, J. Schoenwaelder, and A. B. Ed., "Network Configuration Protocol (NETCONF)," Internet Requests for Comments, RFC Editor, RFC 6241, June 2011. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6241.txt
14. J. Case, M. Fedor, M. Schoffstall, and J. Davin, "A Simple Network Management Protocol (SNMP)," Internet Requests for Comments, RFC Editor, RFC 6241, May 1990. [Online]. Available: http://www.ietf.org/rfc/rfc1157.txt
15. A. Császár, W. John, M. Kind, C. Meirosu, G. Pongrácz, D. Staessens, A. Takács, and J. Westphal, "Unifying cloud and carrier network," *Proceedings of. DCC, Dresden, Germany, to appear Dec*, 2013.
16. W. Shen, M. Yoshida, K. Minato, and W. Imajuku, "vConductor: An enabler for achieving virtual network integration as a service," *Communications Magazine, IEEE*, vol. 53, no. 2, pp. 116–124, 2015.
17. A. Csoma, B. Sonkoly, L. Csikor, F. Németh, A. Gulyas, W. Tavernier, and S. Sahhaf, "Escape: Extensible service chain prototyping environment using mininet, click, netconf and pox," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 125–126, Aug. 2014. [Online]. Available: http://doi.acm.org/10.1145/2740070.2631448
18. C. Basile, A. Lioy, C. Pitscheider, F. Valenza, and M. Vallini, "A novel approach for integrating security policy enforcement with dynamic network virtualization," in *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*, London, UK, April 2015, pp. 1–5, 10.1109/NETSOFT.2015.7116152.
19. S. Spinoso, M. Leogrande, F. Risso, S. Singh, and R. Sisto, "Automatic configuration of opaque network functions in cms," in *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, ser. UCC '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 750–755. [Online]. Available: http://dx.doi.org/10.1109/UCC.2014.122

20. S. Spinoso, M. Virgilio, W. John, A. Manzalini, G. Marchetto, and R. Sisto, "Formal verification of virtual network function graphs in an sp-devops context," in *Service Oriented and Cloud Computing - 4th European Conference, ESOCC 2015, Taormina, Italy, September 15-17, 2015. Proceedings*, 2015, pp. 253–262. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-24072-5_18
21. A. Panda, O. Lahav, K. J. Argyraki, M. Sagiv, and S. Shenker, "Verifying isolation properties in the presence of middleboxes," *CoRR*, vol. abs/1409.7687, 2014. [Online]. Available: http://arxiv.org/abs/1409.7687
22. H. Xu and D. Xiao, "Data modeling for netconf-based network management: Xml schema or yang," in *Communication Technology, 2008. ICCT 2008. 11th IEEE International Conference on*. IEEE, 2008, pp. 561–564.

Appendix 1: XML Schema as Object Model language

```
<schema>
  <element name="router">
    <complexType>
      ....
      <sequence>
        <element name="interfaces"
                 minOccurs="1" maxOccurs="unbounded">
          <attribute name="name" type="string"/>
          <complexType>
            <sequence>
              <element name="ethernet"
                       minOccurs="1" maxOccurs="unbounded">
                <complexType>
                  <attribute name="name" type="string"/>
                  <attribute name="address" type="tns:ipv4"/>
                  <attribute name="hwid" type="tns:eth"/>
                </complexType>
              </element>
            </sequence>
          </complexType>
          <key name="nameEthernetKey">
            <selector xpath="ethernet"/>
            <field xpath="name"/>
          </key>
        </element>
      </sequence>
    </complexType>
    <key name="nameInterfaceKey">
      <selector xpath="interfaces"/>
      <field xpath="name"/>
    </key>
  </element>
 </schema>
```

Listing 7: A possible Object Model description in XML Schema, equivalent to the one written in YANG and shown in Listing 1.